







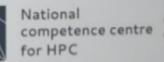






National competence centre for HPC







GPU Programming. When, Why and How?

2024

ENCCS Training





General Considerations



- **Identify Targeted Parts**: The Pareto principle suggests that roughly 10%-20% of the code accounts for 80-90% of the execution time.
- Equivalent GPU Libraries: Check for the equivalent GPU libraries that can replace CPU-based libraries.
- Refactor Loops: Sometimes loops refactoring is necessary to suit the GPU architecture. Often splitting
 the loops is necessary.
- Memory Access Optimization: GPUs perform best when memory access is coalesced and aligned.
 Minimizing global memory accesses and maximizing utilization of shared memory or registers can significantly enhance performance.

Fortran Porting Example. Code I



```
k2 = 0
do i = 1, n_sites
     do j = 1, n_neigh(i)
              k2 = k2 + 1
                counter = 0
                counter2 = 0
               do n = 1, n_max
                     do np = n, n_max
                         do 1 = 0, 1_{max}
                              if( skip_soap_component(1, np, n) )cycle
                               counter = counter+1
                              do m = 0, 1
                                  k = 1 + 1*(1+1)/2 + m
                                   counter2 = counter2 + 1
                                    multiplicity = multiplicity_array(counter2)
                                    soap_rad_der(counter, k2) = soap_rad_der(counter, k2) + multiplicity*real ( <math>cnk_rad_der(k, n, k2)*conjg(cnk(k, np, i)) + cnk(k, n, i)*conjg(cnk(k, np, i)) + cnk(k, np, i)*conjg(cnk(k, np, i))*conjg(cnk(k, np, i))*con
                               end do
                          end do
                      end do
                end do
```

Fortran Porting Example. Code II



False dependencies



The loop starts with:

```
k2 = 0
do i = 1, n_sites
do j = 1, n_neigh(i)
k2 = k2 + 1
```

• Instead one can use:

```
do k2 = 1, k2_max
i=list_of_i(k2)
```

• This allows the parallization over k2 index.

Split the Loop.



Why?

- Registers are limited and the larger the kernel use more registers registers resulting in less active threads (small occupancy).
- In order to compute soap_rad_der(is,k2) the cuda thread needs access to all the previous values soap_rad_der(1:nsoap,k2).
- In order to compute soap_cart_der(1, 1:n_soap, k3) it is required to have access to all values (k3+1:k2+n_neigh(i)).

Part A of the Loop



```
do k2 = 1, k2_{max}
i=list_of_i(k2)
counter = 0
counter2 = 0
do n = 1, n_max
 do np = n, n_max
  do 1 = 0, 1_{max}
   if( skip_soap_component(1, np, n) )cycle
    counter = counter+1
    do m = 0, 1
    k = 1 + 1*(1+1)/2 + m
    counter2 = counter2 + 1
    multiplicity = multiplicity_array(counter2)
     soap_rad_der(counter, k2) = soap_rad_der(counter, k2) + multiplicity*real ( cnk_rad_der(k, n, k2)*conjg(cnk(k, np, i)) + cnk(k, n, i)*conjg (c
  end do
 end do
end do
end do
```

Memory Access Considerations



Consider adjacent threads in the same block and warp.

threadIdx.x	k2	counter	soap_rad_der	cnk_rad_der
0	1	1	0	0
1	2	1	k2_max	nmax * k2_max
2	3	1	2 * k2_max	2 * nmax * k2_max

- The "most" uncoalesced way to access the memory.
- Transposing the data will result in higher performance, despite the extra operations.

Part A with OpenMP Offloading



```
!omp target teams distribute parallel do private(i)
do k2 = 1, k2_max
i=list_of_i(k2)
counter = 0
counter2 = 0
do n = 1, n_max
 do np = n, n_max
  do 1 = 0, 1_max
    if( skip_soap_component(1, np, n) )cycle
    counter = counter+1
    do m = 0, 1
    k = 1 + 1*(1+1)/2 + m
    counter2 = counter2 + 1
    multiplicity = multiplicity_array(counter2)
     tsoap_rad_der(k2,counter) = stoap_rad_der(k2,counter) + multiplicity*real ( tcnk_rad_der(k2,k,n)*conjg(tcnk(i,k,np)) + tcnk(i,k,n)*conjg (tscn
   end do
 end do
end do
end do
```

Part B of the Loop



Assuming the results were (un)transposed.

- We can have each thread computing one k2 but the dot_product(soap(1:n_soap, i), soap_rad_der(1:n_soap, k2)) is computed nsoap times.
- We sacrifice memory for speed and "precompute" dot_product(soap(1:n_soap,
 i), soap_rad_der(1:n_soap, k2))

Part B Splitted



```
do k2 = 1, k2_max
    i=list_of_i(k2)
    locdot=0.d0
    do is=1,nsoap
    locdot=locdot+soap(is, i) * soap_rad_der(is, k2)
    end do
    dot_soap(k2)= locdot
end do
```

Part B with OpenMP Offloading



```
!omp target teams distribute private(i)
do k2 = 1, k2_max
i=list_of_i(k2)
locdot=0.d0
!omp parallel do reduction(+:locdot)
do is=1,nsoap
locdot=locdot+soap(is, i) * soap_rad_der(is, k2)
end do
dot_soap(k2)= locdot
end do
```

```
!omp target teams distribute
do k2 = 1, k2_max
i=list_of_i(k2)

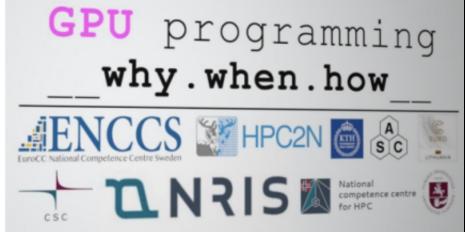
!omp parallel do
do is=1,nsoap
soap_rad_der(is, k2) = soap_rad_der(is, k2) / sqrt_dot_p(i) - soap(is, i) / sqrt_dot_p(i)**3 * dot_soap(k2)
end do
end do
```

- Assuming nsoap ~ 100-1000. The outer loop is distributed over the blocks, while the inner loop over the threads inside of a block.

Part C of the Loop



Part C of the Loop Splitted



```
do i = 1, _sites
    k3=list_k3(i)
    do k2=k3+1,k3+n_neigh(i)
    soap_cart_der(1, 1:n_soap, k3) = soap_cart_der(1, 1:n_soap, k3) - soap_cart_der(1, 1:n_soap, k2)
    end do
end do
```

Part C of the Loop with OpenMP Offloading



```
!omp teams distribute private(k3)
do i = 1, _sites
k3=list_k3(i)

!omp parallel do private(is, k2)
do is=1,n_soap
do k2=k3+1,k3+n_neigh(i)
    soap_cart_der(1, is, k3) = soap_cart_der(1, is, k3) - soap_cart_der(1, is, k2)
    end do
end do
end do
```

Summary



- Find the intensive parts for porting.
- Try to use libraries whenever possible.
- Consider refactoring of the loops.
- Check the memory access patterns
 - on CPU adjacent locations of memory need to be accessed close in time.
 - on GPU adjacent locations of memory need to be accessed by adjacent threads.
- Real life example of porting.